Safe Recursive Boxes

Gérard Boudol

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Safe Recursive Boxes

Gérard Boudol

N° 5115

February 2004

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--5115--FR+ENG

INRIA
SOPHIA ANTIPOLIS

Safe Recursive Boxes

Gérard Boudol

Thème 1 — Réseaux et systèmes
Projets Mimosa

Rapport de recherche n° 5115 — February 2004 — 14 pages

Abstract: We study recursion in call-by-value functional languages, and more specifically a recursion construct recently introduced by Dreyer, which builds recursive boxed values. We design a type and effect system, featuring binary effects, for a call-by-value λ-calculus extended with this recursion construct. We show that this system has the good properties required for an implicitly, statically typed language: the typable expressions do not yield run-time errors, and a principal type can be computed for any typable expression.

Key-words: functional languages, call-by-value, recursion, typing

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Boites Récursives Sûres

Résumé : Nous étudions la récursion dans les langages fonctionnels en appel par valeur, et plus spécifiquement une construction de récursion récemment introduite par Dreyer, qui construit des valeurs récursives allouées. Nous introduisons un système de types et effets, reposant sur des effets binaires, pour un λ-calcul en appel par valeur étendu avec cette construction pour la récursion. Nous montrons que ce système a les bonnes propriétés requises pour un typage statique implicite : les expressions typables ne conduisent pas à des erreurs à l'exécution, et un type principal peut être calculé pour chaque expression typable.

Mots-clés : langages fonctionnels, appel par valeur, récursion, typage

1. Introduction

In this paper we investigate some aspects of recursive definitions in call-by-value languages. It is well-known that in such languages, one is not allowed to use the standard mathematical definition of the fixpoint of a function f, that is, using standard programming language notations:

$$\mathsf{let\ rec}\ x = (fx)\ \mathsf{in}\ \cdots \qquad (\heartsuit)$$

In SML for instance [18], one is compelled to only define recursive functions, and more precisely one may only write

$$\mathsf{let\ rec}\ f = \lambda x M\ \mathsf{in}\ \cdots$$

OCaml [16] is slightly more permissive, since it allows one to use limited forms of constructor applications as the body of recursive definitions. This has been recently extended by Hirschowitz & al. [12], but not to the extent that we can use ($\heartsuit$). The usual call-by-value fixpoint combinator Y, which is such that $(\mathsf{Y}V)$ evaluates into $V(\lambda x.\mathsf{Y}Vx)$, where V is a value, provides a limited form of the equation ($\heartsuit$): we may regard $\mathsf{rec}\ x = (fx)$ as an abbreviation for $x = f(\lambda z.\mathsf{Y}fz)$, but in this way we can only define recursive functions, since $x = \lambda z.\mathsf{Y}fz$, but not recursive values of any kind. In Scheme, one may write

$$\mathsf{let\ rec}\ x = X\ \mathsf{in}\ \cdots$$

for any X, but this is subject to an important restriction: "*it must be possible to evaluate X without assigning or refering to the value of x*" [13]. This actually refers to Landin's implementation of recursion [14], where the recursive variable is interpreted as the value of a fresh memory cell, which can be assigned to and dereferenced. In Scheme, the latter operation is implicit, as all variables are, by default, mutable. If we adopt the more explicit syntax of SML, Landin's interpretation of a recursive definition $\mathsf{rec}\ x = X$ can be described as follows:

$$\begin{array}{rcl} x & = & \mathsf{let}\ r = \mathsf{ref}(\mathsf{raise}\ \mathit{Undefined}) \\ & & \mathsf{in}\ r := X^*\ ;\ (!\, r) \end{array} \qquad (\spadesuit)$$

where X^* is obtained from X by replacing every occurrence of x by an explicit dereferencing $(!\, r)$ of the associated cell. (Notice that we cannot define X^* as $(\lambda x X)(!\, r)$ since $r := X^*$ would then always fail.) This semantics of recursion still does not let us use the fixpoint equation ($\heartsuit$), since its body is translated into $f(!\, r)$, which always raises an exception (unless the computation of f does not terminate). As a matter of fact, it is not obvious to design a semantics of recursion that allows us to use ($\heartsuit$). This would require implementing, by means of a static analysis for instance, the restriction identified in [13], or a conservative approximation of it, in order to obtain a safe construct, that does not introduce run-time errors. The lack of such a semantics and static analysis explains why recursion is usually syntactically restricted in call-by-value languages.

Nevertheless, there has been recently some interest in extending recursion in functional programming languages, with various motivations. For instance, Launchbury & al. [8, 15] point out the need for an extended fixpoint to model circuit feedback in Haskell with monads ([1]). Another major motivation is in the need for mutually recursive modules [1, 5, 7, 9, 11, 20]. Similarly, trying to extend Cardelli's recursive record semantics of objects [4, 21] to objects with state also calls for a general fixpoint [2]. In most of these works, a common issue that has to be investigated is how recursion should interact with computational effects, a topic which is addressed in [8, 10, 19]. To illustrate this point, let us see an example. Following the fixpoint approach to objects, we may define for instance a point object as follows:

$$\begin{array}{rcl} \mathit{point} & = & \mathsf{let\ rec}\ \mathit{self} = \{\mathit{pos} = \mathsf{ref}\ 0, \\ & & \qquad\qquad \mathit{move} = \lambda d(\mathit{self}.\mathit{pos} := !\mathit{self}.\mathit{pos} + d)\} \\ & & \mathsf{in}\ \mathit{self} \end{array}$$

In this particular case we do not really need recursion, since the effect of creating the initial state of the *point* could be lifted out of the scope of the recursion. However, this would be inappropriate for the purpose of inheritance: imagine for instance that we wish to abstract away the particular initial position of a point, by

([1]) In a call-by-name language, the standard (call-by-name) fixpoint combinator Y, which is such that $(\mathsf{Y}f)$ evaluates into $f(\mathsf{Y}f)$, is inappropriate if computing its argument f is intended to have some "global" effect, including all recursive calls in its scope.

defining a class *Point* of points, and then inherit from this class, defining a subclass featuring a new method that resets to a given value the position of the object. Then this new method must have access to the *pos* parameter of the object, which therefore must be available for inheritance. The *self* parameter is introduced exactly for this purpose, of providing access to the components of the object. Then the class of points may be written

$$\begin{aligned} Point \quad = \quad & \lambda x \lambda self\{pos = \mathsf{ref}\ x \\ & \qquad move = \lambda d(self.pos := !self.pos + d)\} \end{aligned}$$

and instantiated like this:

$$point \quad = \quad (\mathsf{let\ rec}\ p = (Point\ 42)p\ \mathsf{in}\ p)$$

As one can see, we are using here an instance of the fixpoint equation (♡), where the evaluation of the function (*Point* 42) from which we are taking the fixpoint yields some computational effects, and where the fixpoint is not a function ([2]). In a recent paper Dreyer [6] points out another situation where computational effects cannot be lifted out of the scope of recursion, namely the separate compilation of mutually recursive modules. Moreover, as shown in [10, 19], one cannot lift effects out of recursion in a language with first class continuations.

In [2] we introduced a recursion construct that allows us to use the fixpoint equation (♡), and to define recursive values whose computation involves computational effects. We introduced a type system to rule out potentially unsafe recursion: in order for $\mathsf{rec}\ x = fx$ to be acceptable, the function f has to be "protective", that is it should not require to know the exact value of its argument. This was applied in particular to the modelling of objects as recursive records. We also designed and proved correct an abstract machine for recursion [3], which is a refinement of Landin's interpretation. However, this approach to generalized recursion has some flaws: first, the type system is not as expressive as one could wish. For instance, if the recursion variable f occurs in M, the recursion $\mathsf{rec}\ f = (\lambda x \lambda y M)N$ is always rejected. Second, the intuitive meaning of the "safeness degrees" introduced in the type system of [2] is not easy to understand (and the proof of type safety is not obvious). Third, the implementation is slightly less efficient than Landin's one.

In this paper we study another proposal, due to Dreyer [6], which may be explained as follows: the translation (♠) introduces a recursive reference r, and transforms the recursive variable into $(!\,r)$. Then one could actually gain something in considering a fixpoint $\mathsf{rec}\ x = X$ where x is a reference. Namely, we gain, with respect to Landin's semantics, the ability to use the recursive variable x as a L-value (i.e. as denoting a memory location) in some cases – typically, when x is used as an argument, like in fx –, whereas we should explicitly dereference it when we really need its R-value – typically, when it is applied to some argument, or, as in the *point* example, when one needs to select one of its components. In fact, the recursive reference is intended to be a *boxed value*, that is a pointer to a heap-allocated box that is assigned once, but not further mutated (although this is still possible in a language with first-class continuations, see [10]). Then we introduce a specific type construct $\tau\,\mathsf{rbox}$ for "recursive boxed values" of type τ. The semantics of recursion is then obvious: like with the ref construct, create a fresh location ℓ as the value of x, evaluate X to a value V, if any, and assign V to ℓ. This is very close to the semantics we had in [2, 3], except that using the R-value of the recursive variable now involves explicitly dereferencing it in X, by means of an operation that we denote unfold. It is immediate to adapt the encoding of objects of [2] to this setting. For instance, the *Point* class above becomes

$$\begin{aligned} Point \quad = \quad & \lambda x \lambda self\{pos = \mathsf{ref}\ x \\ & \qquad move = \lambda d((\mathsf{unfold}\ self).pos := !(\mathsf{unfold}\ self).pos + d)\} \end{aligned}$$

The restriction one has to impose for recursion to be safe does not magically disappear: it is still an error to unfold the value of a recursive variable while computing it. In order to implement this restriction, Dreyer [6] introduces a type system, that we shall discuss below. Our main contribution in this paper is to design a simple type system for safe boxed recursion, adapting the "safeness degrees" of [2]. This will allows us to use the fixpoint equation (♡), provided that the function f is *protective*, which means that it does not require to unfold its argument. For instance (*Point* 42) is a protective function of *self*. We show that our type system enjoys the good properties of ML-like type systems: besides type safety, which is easy to prove, we show that a principal type can be computed for any typable expression.

([2]) There is another possible scenario for objects and inheritance, based on the self-application semantics, that does not use explicit recursion. For a discussion, see [2] for instance.

2. The Calculus

The calculus we study is a call-by-value λ-calculus extended with recursion. Although the need to deal with computational effects in recursive values is one of the main motivations for a general recursion construct, as we saw above, our calculus does not include any effect, apart from recursion itself, nor any other construct like records, pairs, etc. This is only to simplify the presentation: it is very easy to extend the calculus with various other constructs (see [6]). In this paper we shall denote by ϱxM the recursion construct, which is another notation, with a specific semantics, for what we wrote (let rec $x = M$ in x) in the Introduction. Then the syntax of our calculus is:

$$M,\ N\ldots \ ::=\ x \mid \lambda xM \mid \mathsf{unfold} \mid \varrho xM \mid (MN)$$

For instance, one may define the standard mathematical fixpoint combinator:

$$\mathsf{fix} = \lambda f \varrho y (fy)$$

One may also define the call-by-value fixpoint combinator:

$$\mathsf{Y} = \varrho y \lambda f. f \lambda x. (\mathsf{unfold}\, y) f x$$

although we cannot claim this is the right implementation to use (for a typing reason that we shall see later). In comparison with the calculus of Dreyer [6], we observe that we do not use "names", nor types in the syntax: in Dreyer's calculus, recursion is written $\mathsf{rec}(X \triangleright x : \tau.M)$, where X is a (bound) name and τ a type, and there are additional constructs, namely $\lambda X.M$ and (MS) for respectively abstracting a name, and applying an expression to a set S of names (to instantiate the types of recursive variables, that may contain free names). For instance, the fix combinator would be written $\lambda f \mathsf{rec}(X \triangleright x : \tau. f\{X\}x)$ using Dreyer's syntax. We will design a simple type system for the calculus, and we do not need "names". Another minor difference with Dreyer's calculus is that we denote by unfold his unbox operation, and that we do not consider the box operation: here boxing is only performed by recursion.

To describe the operational semantics of the language, and more specifically of recursion, we assume given a set of *locations*, or "boxes", that are pointers to values stored in a heap-allocated memory. Locations are values in the run-time language, the syntax of which may be described as follows:

$$\begin{array}{rcll} V,\ W\ldots & ::= & \ell \mid \lambda xM \mid \mathsf{unfold} & \textit{values} \\ M,\ N\ldots & ::= & x \mid V \mid \varrho xM \mid (MN) & \textit{expressions} \\ P,\ Q\ldots & ::= & M \mid ((\mathsf{set}\,\ell)P) \mid (PN) \mid (VQ) & \textit{code} \end{array}$$

where ℓ is any location. We still use M, $N\ldots$ to denote expressions, even though they may now contain locations. We denote by $\mathsf{loc}(P)$ the set of locations occurring in P, and by $\{x{\mapsto}V\}M$ the capture avoiding substitution of x by V in M.

The operational semantics of the language consists in a transition relation between *configurations*, that are pairs (S, P) of a store S and a code P, to be evaluated. The *store* (or heap) S is a mapping from a finite set, denoted $\mathsf{dom}(S)$, of locations to values. We denote by $S[\ell := V]$ the store, with domain $\mathsf{dom}(S) \cup \{\ell\}$, obtained from S by assigning the value V to ℓ, that is:

$$S[\ell := V](\ell') \ = \ \begin{cases} V & \text{if } \ell' = \ell \\ S(\ell') & \text{otherwise} \end{cases}$$

To describe the transitions, it is convenient to introduce, as usual, *evaluation contexts* (or stacks):

$$\mathbf{E} \ ::= \ [] \mid (\mathbf{E}N) \mid (V\mathbf{E}) \mid ((\mathsf{set}\,\ell)\mathbf{E})$$

Notice that an evaluation context does not contain any binder. Notice also that these contexts correspond to a left-to-right call-by-value strategy (there is no sequencing construct in the language, other than application).

Then the transition rules are:

$$\begin{array}{rcll}
(S, \mathbf{E}[(\lambda x M V)]) & \to & (S, \mathbf{E}[\{x{\mapsto}V\}M]) & \\
(S, \mathbf{E}[(\mathsf{unfold}\,\ell)]) & \to & (S, \mathbf{E}[V]) & S(\ell) = V \\
(S, \mathbf{E}[\varrho x M]) & \to & (S, \mathbf{E}[((\mathsf{set}\,\ell)\{x{\mapsto}\ell\}M)]) & \ell \notin \mathsf{dom}(S) \cup \mathsf{loc}(\mathbf{E}[\varrho x M]) \\
(S, \mathbf{E}[((\mathsf{set}\,\ell)V)]) & \to & (S[\ell := V], \mathbf{E}[V]) &
\end{array}$$

As we said above informally, the evaluation of a recursion $\varrho x M$ consists in creating a (fresh) location ℓ which is the value of the recursive variable (this is expressed by the substitution $\{x{\mapsto}\ell\}$), and then evaluating M. Then $(\mathsf{set}\,\ell)$ is a mark in the evaluation context, to record which location is assigned to when the evaluation of $\{x{\mapsto}\ell\}M$ results in a value V. From the point of view of efficiency, this semantics is similar to the one of Landin, and is slightly better than the one of [3]: since unfolding a recursive variable is explicit, we do not have here to perform a test for definedness of the associated pointer. To evaluate an expression M, we start with the initial configuration $(\emptyset, M)$. The evaluation of M gets stuck if there is a sequence of transitions from the initial configuration to a *faulty* configuration, that is:

$$(\emptyset, M) \xrightarrow{*} (S, \mathbf{E}[(\mathsf{unfold}\,\ell)]) \qquad \text{where} \quad \ell \notin \mathsf{dom}(S)$$

For instance, this is the case with $(\mathsf{fix}\ \mathsf{unfold})$. The task of the type system will be to rule out expressions whose evaluation may result in such a run-time error. In order to prove the type safety property, the following result will be useful. It states that if a location ℓ has been created, from the evaluation of a recursion, then the mark $(\mathsf{set}\,\ell)$ remains in the evaluation context as long as no value has been assigned to ℓ.

LEMMA 2.1. *The property of a configuration* (S, P)

$$\ell \in \mathsf{loc}(P) - \mathsf{dom}(S) \;\Rightarrow\; \exists \mathbf{E}, \exists Q.\ P = \mathbf{E}[((\mathsf{set}\,\ell)Q)]$$

is preserved by the transitions.

PROOF: assume that (S, P) satisfies this property, and that $(S, P) \to (S', P')$ with $\ell \in \mathsf{loc}(P') - \mathsf{dom}(S')$. We proceed by a case analysis. If $P = \mathbf{E}[(\lambda x M V)]$, $S' = S$ and $P' = \mathbf{E}[\{x{\mapsto}V\}M]$, then $\mathsf{loc}(P') \subseteq \mathsf{loc}(P)$, and therefore $P = \mathbf{E}'[((\mathsf{set}\,\ell)R)]$. Since $(\mathsf{set}\,\ell)$ cannot occur in $(\lambda x M V)$, we have $\mathbf{E} = \mathbf{E}'[((\mathsf{set}\,\ell)\mathbf{E}'')]$. The other cases are similar. ❑

COROLLARY 2.2. *Let* M *be an expression such that* $\mathsf{loc}(M) = \emptyset$. *If* $(\emptyset, M) \xrightarrow{*} (S, P)$ *and* $\ell \in \mathsf{loc}(P) - \mathsf{dom}(S)$ *then* $P = \mathbf{E}[((\mathsf{set}\,\ell)Q)]$ *for some* Q.

3. The Type System

Dreyer introduces in [6] a type system, which is in fact a type and effect system, with only one kind of effect, namely the dereferencing of a recursive variable. In order to introduce our own type system, let us review a simplified form of Dreyer's one. The "names" used by Dreyer to record the creation of a location can be seen as *regions*, that are used in effect systems to handle areas in the store [17]. Since a location is associated, at run-time, with a recursive variable, the type of such a variable is decorated by a region, in which the location is supposed to be created. Then the *effect* of a program is a set of regions, denoted E, $F \ldots$, which represents the areas in the store in which the program may perform an unfolding. This is recorded in the types of functions, as the "latent effect" a function may have, when applied to an argument. That is, the types are:

$$\tau,\ \sigma \ldots \ ::= \ \ t \ \mid\ \tau\,\mathsf{rbox}_\rho \ \mid\ (\tau \xrightarrow{E} \sigma)$$

where t is any type variable, and ρ is a region (a name). In Dreyer's system, there is an additional construction, namely universal quantification over a region $\forall \rho.\tau$. The judgements of the simplified version of Dreyer's type system have the form $\Gamma \vdash M : \tau, E$ where Γ is the typing context, the expression M is the subject of the judgement, and τ is its type, while E is its effect. Given the intuitive meaning of regions and effects, the rules of the type system should be clear:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau, \emptyset} \qquad \frac{\Gamma, x : \tau \vdash M : \sigma, E}{\Gamma \vdash \lambda x M : (\tau \xrightarrow{E} \sigma), \emptyset} \qquad \frac{}{\Gamma \vdash \mathsf{unfold} : (\tau\,\mathsf{rbox}_\rho \xrightarrow{\{\rho\}} \tau), \emptyset}$$

$$\frac{\Gamma, x : \tau\,\mathsf{rbox}_\rho \vdash M : \tau, E}{\Gamma \vdash \varrho x M : \tau, E}\ \rho \notin E \qquad \frac{\Gamma \vdash M : (\tau \xrightarrow{E} \sigma), F \qquad \Gamma \vdash N : \tau, G}{\Gamma \vdash (MN) : \sigma, E \cup F \cup G}$$

We notice that a value has an empty effect, and that the only operation that introduces an effect is [the application of] the unfold function. The type of an abstraction $\lambda x M$ records, as we said, the effect of M. In the rule for recursion, the side condition is the one that ensures type safety: it states that the body M of the recursion should not have the effect of unfolding in the region ρ where the location associated with the recursive variable x is supposed to be created. Finally, an application (MN) has all the effects of its components, as well as the latent effect of the function M. For instance, we have, regarding the fix combinator:

$$\vdash \lambda f \varrho y (f y) : (\tau\,\mathsf{rbox}_\rho \xrightarrow{E} \tau) \xrightarrow{E} \tau, \emptyset \qquad \text{provided that} \quad \rho \notin E$$

and therefore one can see that (fix unfold) cannot be typed. In Dreyer's system, the type of fix – that is, $\lambda f \mathsf{rec}(X \triangleright x : \tau . f\{X\}x)$ – is $(\forall X.\mathsf{box}_X(\tau) \xrightarrow{S} \tau) \xrightarrow{S} \tau$, where $X \notin S$.

The type and effect system just described is quite simple and effective: it ensures type safety without, as it seems, rejecting too much expressions – in particular it accepts the fixpoint equation ($\heartsuit$), with some conditions on the function from which we take the fixpoint. However, because of the negative side condition in the typing rule for recursion, it is not very clear how well it supports the implicit typing discipline. That is, can we compute a type for any typable expression? What could be the notion of a principal type? Looking at the type of fix for instance, it seems that one should incorporate quantification in types, as Dreyer does, but it is not clear that the resulting system supports type inference (see [6] for a discussion). We shall not investigate these questions here. Instead, we shall derive from the type and effect system above a simpler system, which is less expressive but enjoys the good properties required for an implicitly, statically typed language.

The idea, inspired by the notion of a "safeness degree" from [2], is to consider a *binary* (or boolean) version of the effects: the effects are reduced to 1 – good, no effect at all – and 0 – bad, maybe some effect. In this setting, the only possibility to express the side condition on typing recursion is to assume that the body of the recursion has an empty set of effects, that is, it has "degree" 1. (We may also distinguish a special case, where the recursive variable does not occur free in the body of the recursion.) In this way, we no longer need the regions. It should be clear how to obtain the rules of the "binary effects" system from the previous ones: an effect which is known to be non-empty is translated to 0, whereas an effect which is known to be empty, or is required not to contain some effect is approximated as 1. Then, if we denote by α, $\beta \ldots$ binary effects, and by $\alpha \wedge \beta$ their conjunction, corresponding to set union, with $1 \wedge 1 = 1$, $0 \wedge 1 = 0$, etc., the rules are:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau, 1} \qquad \frac{\Gamma, x : \tau \vdash M : \sigma, \alpha}{\Gamma \vdash \lambda x M : (\tau \xrightarrow{\alpha} \sigma), 1} \qquad \frac{}{\Gamma \vdash \mathsf{unfold} : (\tau\,\mathsf{rbox} \xrightarrow{0} \tau), 1}$$

$$\frac{\Gamma, x : \tau\,\mathsf{rbox} \vdash M : \tau, 1}{\Gamma \vdash \varrho x M : \tau, 1} \qquad \frac{\Gamma \vdash M : (\tau \xrightarrow{\alpha} \sigma), \beta \qquad \Gamma \vdash N : \tau, \gamma}{\Gamma \vdash (MN) : \sigma, \alpha \wedge \beta \wedge \gamma}$$

For instance, we have

$$\vdash \lambda f \varrho y (f y) : (\tau\,\mathsf{rbox} \xrightarrow{1} \tau) \xrightarrow{1} \tau, 1$$

and (fix unfold) is still not typable. A function of type $(\theta \xrightarrow{1} \sigma)$, with no latent effect, may be called "protective", as in [2], and the typing of fix says that one can safely take the fixpoint of a protective function – a sensible restriction on the fixpoint equation ($\heartsuit$). Since there is only one region in the binary effects system, it should be clear that this system cannot be as precise as the effect system. For instance, the expression $\varrho y(\mathsf{unfold}(\mathsf{K}xy))$, where $\mathsf{K} = \lambda x \lambda y\, x$, is not typable with binary effects, whereas it is accepted in the full effect system, provided that the variables x and y have types annotated with distinct regions. A more serious example is the following: we may define SML recursion $\mathsf{rec}\ f = \lambda x M$ as $f = \varrho f \lambda x M$, or equivalently as $f = (\mathsf{Y}\,\lambda f \lambda x M)$. However, if M contains recursive calls to f which are not "protected" – which is generally the case –, that is if M has effect 0, then any application (fN) will have effect 0, and this is unfortunate since we then cannot use such an expression unprotected in another recursion. Therefore the recursion construct $\varrho x M$ together with the binary effect system is not very well-suited for dealing with recursive functions. On

$$\frac{C\,;\Gamma\vdash M:\tau,\alpha \qquad C\vdash\beta\leqslant\alpha}{C\,;\Gamma\vdash M:\tau,\beta} \qquad \frac{}{C\,;\Gamma,x:\tau\vdash x:\tau,1}$$

$$\frac{C\,;\Gamma,x:\tau\vdash M:\sigma,a}{C\,;\Gamma\vdash\lambda xM:(\tau\xrightarrow{a}\sigma),1} \qquad \frac{C\,;\Gamma\vdash M:(\tau\xrightarrow{a}\sigma),\alpha \qquad C\,;\Gamma\vdash N:\tau,\beta}{C\,;\Gamma\vdash(MN):\sigma,a\wedge\alpha\wedge\beta}$$

$$\frac{}{C\,;\Gamma\vdash \mathsf{unfold}:(\tau\,\mathsf{rbox}\xrightarrow{0}\tau),1} \qquad \frac{C\,;\Gamma,x:\tau\,\mathsf{rbox}\vdash M:\tau,1}{C\,;\Gamma\vdash\varrho xM:\tau,1}$$

Figure 1: The Type and Binary Effect System

the other hand, the binary effect system accepts expressions that could not be typed with the system of [2]. For instance, $\varrho f.(\lambda x\lambda y M)N$ is accepted, provided that N has no effect (and appropriate typing). Moreover, the intuitive meaning of binary effects is more clear than the one of the "safeness degrees". In retrospect, the type system of [2] appears to be an attempt to build a type and effect system in a "hostile context", where the effects were hidden from the syntax, and could only be detected through evaluation contexts, like $([]\,V)$ or $[].m$. A disavantage of explicitly boxed recursive values, with respect to the fixpoint used in [2], is that one has to anticipate a type of the form $\tau\,\mathsf{rbox}$ for an argument that will be used as a fixpoint. Typically, we cannot apply the fix combinator to a function of type $(\tau\xrightarrow{\alpha}\tau)$, as one could wish to do([3]). Therefore the construct ϱxM, while being adequate for specific purposes, like objects as recursive records, may not actually qualify as a general recursive construct. We may still need more traditional forms of recursion, like the one investigated in [12].

The type and (binary) effect system we shall formally study here is slightly more elaborated than the one we just introduced: in order to perform type inference, we need effect variables. For instance, trying to build a type for (fx), we see that f must have a type of the form $(\tau\xrightarrow{\alpha}\sigma)$, but there is no constraint imposed on α, which should then be an effect variable. In order to have only simple constraints to solve in inferring types, we distinguish "effects", that are either constants 0 or 1, or variables, from "effect expressions", that are build from effects using the conjunction operation. Effect expressions are not supposed to occur in the arrow types (but this is not a serious restriction). Then, assuming given a denumerable set $\mathcal{EV}ar$ of effect variables, the syntax of types and effects is as follows:

$p,\ q\ldots$	$\in$	$\mathcal{EV}ar$	*effect variables*
$a,\ b,\ c\ldots$	$\in$	$\{0,1\}\cup\mathcal{EV}ar$	*effects*
$\alpha,\ \beta,\ \gamma\ldots$	::=	$a \mid (\alpha\wedge\beta)$	*effect expressions*
$\tau,\ \sigma\ldots$	::=	$t \mid (\tau\xrightarrow{a}\sigma) \mid \tau\,\mathsf{rbox}$	*types*

The typing judgements have the form $C\,;\Gamma\vdash M:\tau,\alpha$, where C is a *constraint*, that is a finite set of inequations $p_1\leqslant\alpha_1,\ldots,p_n\leqslant\alpha_n$. Notice that such a constraint is always satisfiable, by assigning 0 to every effect variable for instance. In the typing rule we use judgements of the form $C\vdash\beta\leqslant\alpha$, which are proved by means of an inference system which should be obvious, namely:

$$\frac{}{C\vdash 0\leqslant\alpha} \qquad \frac{}{C\vdash\alpha\leqslant 1} \qquad \frac{}{C\vdash\alpha\leqslant\alpha} \qquad \frac{C\vdash\alpha\leqslant\gamma \quad C\vdash\gamma\leqslant\beta}{C\vdash\alpha\leqslant\beta}$$

$$\frac{}{p\leqslant\alpha\,,\,C\vdash p\leqslant\alpha} \qquad \frac{}{C\vdash(\alpha\wedge\beta)\leqslant\alpha} \qquad \frac{}{C\vdash(\alpha\wedge\beta)\leqslant\beta} \qquad \frac{C\vdash\gamma\leqslant\alpha \quad C\vdash\gamma\leqslant\beta}{C\vdash\gamma\leqslant(\alpha\wedge\beta)}$$

Then the typing system is given in Figure 1. The first rule is an effect weakening rule, stating that a "good" effect can always be safely downgraded. In the rule for typing abstractions, the restriction that only effects can occur in types is not very serious: if we can infer $C\,;\Gamma,x:\tau\vdash M:\sigma,\alpha$, we can also infer $p\leqslant\alpha,C\,;\Gamma,x:\tau\vdash M:\sigma,p$ where p is a fresh effect variable, thanks to the effect weakening rule, and we

([3]) One can tranform, by a kind of η-expansion, an expression M of this type into an expression of type $(\tau\,\mathsf{rbox}\xrightarrow{0}\tau)$, namely $\lambda x.M(\mathsf{unfold}\ x)$, but fix cannot be applied to the latter, which is not protective.

may now apply the rule for typing abstractions. We could also have restricted effect weakening to take place only when typing an abstraction:

$$\frac{C\,;\,\Gamma, x:\tau \vdash M:\sigma,\alpha \qquad C \vdash a \leqslant \alpha}{C\,;\,\Gamma \vdash \lambda x M : (\tau \xrightarrow{a} \sigma), 1}$$

4. Type Safety

In order to prove type safety, we first extend the type system to the run-time language. To this end, we assume that the typing contexts contain not only assumptions about the type of variables, but also similar assumptions about locations. Then we have the rules:

$$\frac{}{C\,;\,\Gamma, \ell:\tau \vdash \ell : \tau\,\mathsf{rbox}, 1} \qquad \frac{C\,;\,\Gamma, \ell:\tau \vdash P:\tau, 1}{C\,;\,\Gamma, \ell:\tau \vdash ((\mathsf{set}\,\ell)P):\tau, 1}$$

The rules for typing (PN) and (VQ) are the same as the one for (MN), and are omitted.

A configuration is faulty if the code to be computed calls for a value of a location ℓ from the store S, but this location has no value in S. That is, a *faulty* configuration has the form $(S, \mathbf{E}[(\mathsf{unfold}\,\ell)])$ with $\ell \notin \mathsf{dom}(S)$. By the Corollary 2.2, if such a configuration is reached during the evaluation of an expression, that is

$$(\emptyset, M) \xrightarrow{*} (S, \mathbf{E}[(\mathsf{unfold}\,\ell)]) \qquad \text{with} \quad \ell \notin \mathsf{dom}(S)$$

we have $\mathbf{E} = \mathbf{E}_0[((\mathsf{set}\,\ell)\mathbf{E}_1)]$. It is easy to check that such an expression is not typable:

LEMMA 4.1. *The expressions of the form* $\mathbf{E}_0[((\mathsf{set}\,\ell)\mathbf{E}_1[(\mathsf{unfold}\,\ell)])]$ *are not typable.*

PROOF: the expression $(\mathsf{unfold}\,\ell)$ has effect 0. More precisely, if $C\,;\,\Gamma, \ell:\tau \vdash (\mathsf{unfold}\,\ell):\tau,\alpha$ is a valid judgement, then we have $C \vdash \alpha \leqslant 0$. It is easy to check, by induction on the structure, that for any evaluation context $\mathbf{E}$, if M has effect 0, in the previous sense, then $\mathbf{E}[M]$ also has effect 0, or is not typable. Then $((\mathsf{set}\,\ell)\mathbf{E}_1[(\mathsf{unfold}\,\ell)])$ is not typable, since set requires its second argument to have effect 1. ❑

Then to establish type safety, it only remains to show the standard subject reduction property. To this end we prove, as usual, a substitution lemma:

LEMMA 4.2. *If* $C\,;\,\Gamma, x:\tau \vdash M:\sigma,\alpha$ *and* $C\,;\,\Gamma \vdash V:\tau,\beta$ *then* $C\,;\,\Gamma \vdash \{x{\mapsto}V\}M:\sigma,\alpha$.

PROOF: it is easy to see that values have effect 1, that is, if $C\,;\,\Gamma \vdash V:\tau,\beta$ is a valid judgement, then $C\,;\,\Gamma \vdash V:\tau,1$ is valid too (by induction on the proof of the first judgement). Then we show the Lemma by induction on the proof of $C\,;\,\Gamma, x:\tau \vdash M:\sigma,\alpha$, and by case on the last rule used in this proof. If this is an axiom, with $M = x$, $\sigma = \tau$ and $\alpha = 1$, then we use the fact that V has effect 1 to conclude. If this is an axiom with $M \neq x$, then we use the fact, which is easy to prove, that if $C\,;\,\Gamma, x:\tau \vdash M:\sigma,\alpha$ and x is not free in M, then $C\,;\,\Gamma \vdash M:\sigma,\alpha$. All the other cases are immediate (modulo α-conversion in the case where M is an abstraction or a recursion). ❑

Now to prove subject reduction, we extend the typing to the store:

$$\frac{}{C\,;\,\Gamma \vdash \emptyset} \qquad \frac{C\,;\,\Gamma, \ell:\tau \vdash S \qquad C\,;\,\Gamma, \ell:\tau \vdash V:\tau,\alpha}{C\,;\,\Gamma, \ell:\tau \vdash S[\ell := V]}$$

and then to configurations:

$$\frac{C\,;\,\Gamma \vdash S \qquad C\,;\,\Gamma \vdash P:\tau,\alpha}{C\,;\,\Gamma \vdash (S,P):\tau}$$

LEMMA (SUBJECT REDUCTION) 4.3. *If* $C\,;\,\Gamma \vdash (S,P):\tau$ *and* $(S,P) \to (S',P')$ *then there exists* Γ' *such that* $C\,;\,\Gamma' \vdash (S',P'):\tau$ *and* $\Gamma' = \Gamma$ *or* $\Gamma' = \Gamma, \ell:\sigma$ *for some* ℓ *and* σ.

PROOF: this is a standard result (*cf.* [22]), and we only sketch the proof. If $(S,P) \to (S',P')$ then $P = \mathbf{E}[R]$ for some evaluation context $\mathbf{E}$ and some redex R, and $P' = \mathbf{E}[Q]$. Then the proof of the judgement $C\,;\Gamma \vdash (S,P) : \tau$ has the following form:

$$\dfrac{\dfrac{\vdots}{C\,;\Gamma \vdash S} \qquad \dfrac{\dfrac{\vdots}{C\,;\Delta \vdash R : \sigma,\beta}}{C\,;\Gamma \vdash \mathbf{E}[R] : \tau,\alpha}}{C\,;\Gamma \vdash (S,\mathbf{E}[R]) : \tau}\ \Pi$$

where we actually have $\Delta = \Gamma$, since $\mathbf{E}$ is an evaluation context, which does not involve any binder. We examine the possible cases for R, and in each case we proceed by induction on the proof of $C\,;\Gamma \vdash R : \sigma,\beta$, and then by cases on the last rule used in the proof, in order to show that this proof can be replaced in Π by a proof of $C\,;\Gamma' \vdash Q : \sigma,\beta$ (and simultaneously the proof of $C\,;\Gamma \vdash S$ can be replaced by a proof of $C\,;\Gamma' \vdash S'$), to finally obtain a proof of $C\,;\Gamma' \vdash (S',\mathbf{E}[Q]) : \tau$. There are two cases: either the last rule in proving $C\,;\Gamma \vdash R : \sigma,\beta$ is the effect weakening rule, in which case we simply use the induction hypothesis, or it is a rule depending on the syntax of the expression.

- If $R = (\lambda x M V)$ and $Q = \{x{\mapsto}V\}M$ then the proof replacement is as follows, thanks to the Lemma 4.2:

$$\dfrac{\dfrac{\dfrac{\vdots}{C\,;\Gamma,x:\theta \vdash M : \sigma,\gamma}}{C\,;\Gamma \vdash \lambda x M : \theta \xrightarrow{a} \sigma,\delta_0} \qquad \dfrac{\vdots}{C\,;\Gamma \vdash V : \theta,\delta_1}}{C\,;\Gamma \vdash (\lambda x M V) : \sigma,\beta} \quad \rightsquigarrow \quad \dfrac{\dfrac{\vdots}{C\,;\Gamma \vdash \{x{\mapsto}V\}M : \sigma,\gamma} \qquad \dfrac{\vdots}{C \vdash \beta \leqslant \gamma}}{C\,;\Gamma \vdash \{x{\mapsto}V\}M : \sigma,\beta}$$

where $C \vdash a \leqslant \gamma$ and $\beta = a \wedge \delta_0 \wedge \delta_1$.

- If $R = (\mathsf{unfold}\,\ell)$ and $Q = V = S(\ell)$, we must have $\Gamma = \Gamma',\ell : \sigma$ (and $C \vdash \beta \leqslant 0$) since $C\,;\Gamma \vdash (\mathsf{unfold}\,\ell) : \sigma,\beta$, and, since $S = S[\ell := V]$, we have $C\,;\Gamma \vdash V : \sigma,\gamma$. We already observed that $C\,;\Gamma \vdash V : \sigma,1$ is therefore valid, hence $C\,;\Gamma \vdash V : \sigma,\beta$ by the effect weakening rule. We then replace the proof of typing of R in Π by a proof of this latter judgement.

- If $R = \varrho x M$ and $Q = ((\mathsf{set}\,\ell)\{x{\mapsto}\ell\}M)$ where $\ell \notin \mathsf{dom}(S) \cup \mathsf{loc}(\mathbf{E}[R])$ then the proof replacement is as follows, thanks to the Lemma 4.2, using a weakening property, namely that one can add hypotheses in the typing context provided that they do not concern [a variable or] a location occurring in the subject of the judgement (this is easy to prove):

$$\dfrac{\dfrac{\vdots}{C\,;\Gamma,x:\sigma\,\mathsf{rbox} \vdash M : \sigma,1}}{C\,;\Gamma \vdash \varrho x M : \sigma,\beta} \quad \rightsquigarrow \quad \dfrac{\dfrac{\vdots}{C\,;\Gamma,\ell:\sigma \vdash \{x{\mapsto}\ell\}M : \sigma,1}}{C\,;\Gamma,\ell:\sigma \vdash ((\mathsf{set}\,\ell)\{x{\mapsto}\ell\}M) : \sigma,\beta}$$

(with $\beta = 1$). One then uses the weakening property to conclude.

- Finally if $R = ((\mathsf{set}\,\ell)V)$ and $Q = V$ and $S' = S[\ell := V]$, we have $\Gamma = \Gamma',\ell : \sigma$, and the replacements to perform in Π are as follows. First, regarding R:

$$\dfrac{\dfrac{\vdots}{C\,;\Gamma \vdash V : \sigma,1}}{C\,;\Gamma \vdash ((\mathsf{set}\,\ell)V) : \sigma,1} \quad \rightsquigarrow \quad \dfrac{\vdots}{C\,;\Gamma \vdash V : \sigma,1}$$

and, regarding S:

$$\frac{\vdots}{C\,;\Gamma\vdash S}\quad\rightsquigarrow\quad\frac{\dfrac{\vdots}{C\,;\Gamma\vdash S}\qquad\dfrac{\vdots}{C\,;\Gamma\vdash V:\sigma,1}}{C\,;\Gamma\vdash S'}$$

This concludes the proof of subject reduction. ❑

Putting together the Lemmas 4.3 and 4.1, we obtain:

PROPOSITION (TYPE SAFETY) 4.4. *Let M be a typable expression of the source language, that is, $\mathsf{loc}(M)=\emptyset$. Then evaluating M does not result in a faulty configuration, that is, if $(\emptyset,M)\xrightarrow{*}(S,P)$ then (S,P) is not faulty.*

Extending the notion of a faulty configuration to deal with the usual type errors, that is $(S,\mathbf{E}[(\ell\,V)])$ and $(S,\mathbf{E}[(\mathsf{unfold}\,V)])$ where V is not a location, we could also prove the usual "progress" property, by showing that for any configuration (S,P) reachable for $(\emptyset,M)$ where M is closed, either P is a value, or $(S,P)\to(S',P')$ for some (S',P'), or (S,P) is faulty.

5. Type Inference

In order to perform type inference, we need to solve type equations, as usual, and this in turn involves solving effect equations. Typically, an equation $(\tau_0\xrightarrow{a}\sigma_0)=(\tau_1\xrightarrow{b}\sigma_1)$ is solved by solving $\tau_0=\tau_1$, $a=b$ and $\sigma_0=\sigma_1$ ([4]). In the type inference process, we shall, in the case of an abstraction, introduce an effect variable with a constraint $p\leqslant\alpha$. Since p may be equated to some effect a, we may have to examine inequations of the form $a\leqslant\alpha$. Therefore, what we have to solve is a finite set A of assertions that are either type equations $\tau=\sigma$, or effect equations $a=b$, or inequalities $a\leqslant\alpha$. A *solution* of A is a pair (C,S) of a constraint C and a type and effect substitution S such that $C\vdash\mathsf{S}A$, that is $C\vdash\mathsf{S}\tau=\mathsf{S}\sigma$ for all type equations $\tau=\sigma$ in A, $C\vdash\mathsf{S}a=\mathsf{S}b$ for all effect equations $a=b$ in A and $C\vdash\mathsf{S}a\leqslant\mathsf{S}\alpha$ for all inequalities $a\leqslant\alpha$ in A. A solution (C,S) of A is *more general* than another one (C',S') if there exists a substitution S'' such that $\mathsf{S}'=\mathsf{S}''\mathsf{S}$ and $C'\vdash\mathsf{S}''C$. As usual, solving a set of assertions will consist in transforming A into an equivalent – that is, having the same solutions – set of assertions B which "is" a solution, or more precisely has the shape of a solution. A set B of assertions is in *solved form* if

$$B=\{t_1=\tau_1,\ldots,t_n=\tau_n\}\cup\{p_1=a_1,\ldots,p_k=a_k\}\cup\{q_1\leqslant\alpha_1,\ldots,q_m\leqslant\alpha_m\}$$

where

(i) the type variable t_i only occurs in B as the left member of the equation $t_i=\tau_i$;

(ii) the effect variable p_j only occurs in B as the left member of the equation $p_j=a_j$.

It should be clear that if B is a solved form, then the pair (C,S) where $C=\{q_1\leqslant\alpha_1,\ldots,q_m\leqslant\alpha_m\}$ and $\mathsf{S}=\{t_1\mapsto\tau_1,\ldots,t_n\mapsto\tau_n\}\cup\{p_1\mapsto a_1,\ldots,p_k\mapsto a_k\}$ is a most general solution of B.

The transformation of sets of assertions consists, as usual, in decomposing assertions and replacing variables by their value. We call this process the decomposition relation $A\rhd A'$. It is described in Figure 2, where e stands for either a type or an effect, and x is a type or effect variable. We write $(C,\mathsf{S})=\mathcal{S}ol(A)$ if (C,S) is the canonical solution associated with a solved form B such that $A\rhd^*B$. This is justified by the following property:

PROPOSITION 5.1.

(i) *The decomposition $\rhd$ terminates.*

(ii) *If $A\rhd A'$ then A and A' have the same solutions.*

(iii) *If A is irreducible with respect to $\rhd$, then A is not a solved form if and only if it contains an equation $\theta\,\mathsf{rbox}=(\tau\xrightarrow{a}\sigma)$, or a symmetric equation, or $t=\tau$ where the variable t occurs in τ, or $1\leqslant 0$, or $0=1$ or the symmetric equation.*

(iv) *If A is irreducible with respect to $\rhd$, then it has a solution if and only if it is a solved form.*

([4]) This explains the restriction we imposed on effects in types: having to solve equations $\alpha=\beta$ on effect expression would certainly be more difficult than solving $a=b$, which is trivial.

$$
\begin{array}{rcll}
\{e = e\} \cup A & \rhd & A & \\
\{e = x\} \cup A & \rhd & \{x = e\} \cup A & e \text{ is not a variable} \\
\{x = e\} \cup A & \rhd & \{x = e\} \cup \{x \mapsto e\}A & x \text{ occurs in } A \text{ and not in } e \\
\{0 \leqslant \alpha\} \cup A & \rhd & A & \\
\{1 \leqslant (\alpha \wedge \beta)\} \cup A & \rhd & \{1 \leqslant \alpha, 1 \leqslant \beta\} \cup A & \\
\{a \leqslant 1\} \cup A & \rhd & A & \\
\{1 \leqslant p\} \cup A & \rhd & \{p = 1\} \cup A & \\
\{(\tau_0 \xrightarrow{a} \sigma_0) = (\tau_1 \xrightarrow{b} \sigma_1)\} \cup A & \rhd & \{\tau_0 = \tau_1, a = b, \sigma_0 = \sigma_1\} \cup A & \\
\{\tau\,\mathsf{rbox} = \sigma\,\mathsf{rbox}\} \cup A & \rhd & \{\tau = \sigma\} \cup A &
\end{array}
$$

Figure 2: Decomposition of Assertions

The proof is similar to the one given in [2], and even simpler since we do not have to take care here about the "well-formedness" of types. Regarding the types, this is actually a standard result. Moreover, the decomposition process regarding effects is indeed very simple.

COROLLARY 5.2. *For any set of assertion A, either A has a most general solution, or the decomposition of A fails, that is $A \rhd^* A'$ where A' is irresolvable (i.e. not decomposable and not a solved form).*

Now we describe the algorithm that tries to build a typing for an expression as a function $\mathsf{Typ}(M)$ which either reports a failure or returns a tuple $(C\,;\Gamma\,;\tau,\alpha)$, that is intended to be a principal typing for M, where M is an expression of the source language, that is $\mathsf{loc}(M) = \emptyset$. Let us fix the terminology here: a typing for M is a tuple $(C\,;\Gamma\,;\tau,\alpha)$ such that $C\,;\Gamma \vdash M : \tau,\alpha$ is provable. This typing is *principal* if for any other typing $(C'\,;\Delta\,;\sigma,\beta)$ for M there exists a type and effect substitution S such that $C' \vdash \mathsf{S}C$, $\mathsf{S}\Gamma \subseteq \Delta$, $\sigma = \mathsf{S}\tau$ and $\beta = \mathsf{S}\alpha$. To give the definition of the typing algorithm, we need some notations: for any typing context Γ, we denote by $\Gamma\backslash x$ the typing context obtained from Γ by removing the assumption it may contain about x. Given two typing contexts Γ and Δ, we denote by $\{\Gamma = \Delta\}$ the set of type equations defined by

$$\{\Gamma = \Delta\} = \{\, \Gamma(x) = \Delta(x) \mid x \in \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) \,\}$$

The function Typ is defined by cases, as follows, where it should be understood that, when a condition $(C, \mathsf{S}) = \mathcal{S}ol(A)$ cannot be met, because A has no solution, then the algorithm reports a failure.

- $\mathsf{Typ}(x) = (\emptyset\,;x : t\,;t,p)$ where t and p are fresh.
- $\mathsf{Typ}(\mathsf{unfold}) = (\emptyset\,;\emptyset\,;t\,\mathsf{rbox} \xrightarrow{0} t,p)$ where t and p are fresh.
- If $\mathsf{Typ}(M) = (C\,;\Gamma\,;\tau,\alpha)$ and $\Gamma(x) = \sigma$ (with the convention that σ is a fresh type variable if $x \notin \mathsf{dom}(\Gamma)$) then $\mathsf{Typ}(\lambda x M) = (\{p \leqslant \alpha\} \cup C\,;\Gamma\backslash x\,;(\sigma \xrightarrow{p} \tau),q)$ where p and q are fresh effect variables.
- If $\mathsf{Typ}(M) = (C\,;\Gamma\,;\tau,\alpha)$ and $\Gamma(x) = \sigma$ (with the same convention as above), and if $(C', \mathsf{S}) = \mathcal{S}ol(C \cup \{1 \leqslant \alpha, \sigma = \tau\,\mathsf{rbox}\})$ then $\mathsf{Typ}(\varrho x M) = (C'\,;\mathsf{S}(\Gamma\backslash x)\,;\mathsf{S}\tau,p)$, where p is a fresh effect variable.
- If $\mathsf{Typ}(M) = (C_0\,;\Gamma_0\,;\tau_0,\alpha_0)$ and $\mathsf{Typ}(N) = (C_1\,;\Gamma_1\,;\tau_1,\alpha_1)$, and if $(C, \mathsf{S}) = \mathcal{S}ol(C_0 \cup C_1 \cup \{\tau_0 = (\tau_1 \xrightarrow{p} t)\} \cup \{\Gamma_0 = \Gamma_1\})$ then $\mathsf{Typ}(MN) = (C\,;\mathsf{S}\Gamma_0 \cup \mathsf{S}\Gamma_1\,;\mathsf{S}t, \mathsf{S}p \wedge \mathsf{S}\alpha_0 \wedge \mathsf{S}\alpha_1)$, where t and p are fresh.

It is easy to see that $\mathsf{Typ}(M)$ always terminate, since it is recursively called on smaller arguments, and the decomposition of assertions terminates. The function Typ reports a failure if one of its recursive calls does, which is the case if the set of assertions associated with a recursion or application node has no solution. For instance, the algorithm fails for $\varrho x(\mathsf{unfold}\,x)$ because the condition $1 \leqslant 0$ cannot be met. It fails for $\varrho x\,x$ because the equation $\sigma = \tau\,\mathsf{rbox}$, and more precisely $t = t\,\mathsf{rbox}$, has no solution in this case. For a similar reason, it fails for (xx) because the condition $\{\Gamma_0 = \Gamma_1\}$ has no solution here. It also fails if a recursive box is used as a function, like in $\varrho x(xy)$, but it succeeds with $\varrho x \lambda y((\mathsf{unfold}\,x)y)$ for instance. The following properties establish the correctness of the algorithm:

PROPOSITION (SOUNDNESS) 5.3. *If $\mathsf{Typ}(M) = (C\,;\Gamma\,;\tau,\alpha)$ then $C\,;\Gamma \vdash M : \tau,\alpha$ is a valid judgement.*

The proof, by induction on M, is standard (in the case of recursion and application, one needs a substitution property, namely that if $C\,;\Gamma \vdash M : \tau, \alpha$ and $C' \vdash \mathsf{S}C$ then $C'\,;\mathsf{S}\Gamma \vdash M : \mathsf{S}\tau, \mathsf{S}\alpha$, which is easy to prove, by induction on the proof of the first judgement).

PROPOSITION (COMPLETENESS) 5.4. *If M is typable then $\mathsf{Typ}(M)$ does not fail, and $(C\,;\Gamma\,;\tau,\alpha) = \mathsf{Typ}(M)$ is a principal typing for M.*

Again, the proof, by induction on the proof of a valid judgement $C'\,;\Delta \vdash M : \sigma, \beta$, is standard. It relies on the fact that $\mathcal{S}ol(A)$, when it exists, is a most general solution of A.

References

[1] D. ANCONA, S. FAGORZI, E. MOGGI, E. ZUCCA, *Mixin modules and computational effects,* ICALP'03, Lecture Notes in Comput. Sci. 2719 (2003) 224-238.

[2] G. BOUDOL, *The recursive record semantics of objects revisited,* INRIA Res. Rep. 4199, to appear in the Journal of Functional Programming (2001).

[3] G. BOUDOL, P. ZIMMER, *Recursion in the call-by-value λ-calculus,* FICS'02 (2002).

[4] L. CARDELLI, *A semantics of multiple inheritance,* Semantics of Data Types, Lecture Notes in Comput. Sci. 173 (1984) 51-67. Also published in Information and Computation, Vol. 76 (1988).

[5] K. CRARY, R. HARPER, S. PURI, *What is a recursive module?,* PLDI'99 (1999) 50-63.

[6] D. DREYER, *A type system for well-founded recursion,* POPL'04 (2004) 293-305.

[7] D. DUGGAN, C. SOURELIS, *Mixin modules,* ICFP'96 (1996) 262-273.

[8] L. ERKÖK, J. LAUNCHBURY, *Recursive monadic bindings,* ICFP'00 (2000) 174-185.

[9] M. FLATT, M. FELLEISEN, *Units: cool modules for HOT languages,* PLDI'98 (1998) 236-248.

[10] D.P. FRIEDMAN, A. SABRY, *Recursion is a computational effect,* Tech. Rep. 546, CS Dept. Indiana University (2000).

[11] T. HIRSCHOWITZ, X. LEROY, *Mixin modules in a call-by-value setting,* ESOP'02, Lecture Notes in Comput. Sci. 2305 (2002) 6-20.

[12] T. HIRSCHOWITZ, X. LEROY, J.B. WELLS, *Compilation of extended recursion in call-by-value functional languages,* PPDP'03 (2003) 160-171.

[13] R. KELSEY, W. CLINGER, J. REEDS (EDS.), *Revised^5 Report on the Algorithmic Language Scheme,* Higher-Order and Symbolic Computation Vol. 11, No. 1 (1998).

[14] P.J. LANDIN, *The mechanical evaluation of expressions,* Computer Journal Vol. 6 (1964) 308-320.

[15] J. LAUNCHBURY, J.R. LEWIS, B. COOK, *On embedding a microarchitectural design within Haskell,* ICFP'99 (1999) 60-69.

[16] X. LEROY, D. DOLIGEZ, J. GARRIGUE, D. RÉMY, J. VOUILLON, *The Objective Caml System, release 3.07,* Documentation and user's manual, available at `http://caml.inria.fr` (2003).

[17] J.M. LUCASSEN, D.K. GIFFORD, *Polymorphic effect systems,* POPL'88 (1988) 47-57.

[18] R. MILNER, M. TOFTE, R. HARPER, D. MACQUEEN, *The definition of Standard ML (Revised),* The MIT Press (1997).

[19] E. MOGGI, A. SABRY, *An abstract monadic semantics for value recursion,* FICS'03 (2003).

[20] C. RUSSO, *Recursive structures for Standard ML,* ICFP'01 (2001) 50-61.

[21] M. WAND, *Type inference for objects with instance variables and inheritance,* in Theoretical Aspects of Object-Oriented Programming (C. Gunter, J. Mitchell Eds.) (1994) 97-120.

[22] A. Wright, M. Felleisen, *A syntactic approach to type soundness*, Information and Computation Vol. 115 No. 1 (1994) 38-94.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399